Exceptions Prove the Rule: Investigating and Resolving Residual Side Channels in Provably Secure Interrupt Handling

Matteo Busi
Università Ca' Foscari, Italy

Pierpaolo Degano
Università di Pisa & IMT Lucca, Italy

Riccardo Focardi
Università Ca' Foscari, Italy

Letterio Galletta
IMT Lucca, Italy

Flaminia Luccio
Università Ca' Foscari, Italy

Frank Piessens
KU Leuven, Belgium

Jo Van Bulck
KU Leuven, Belgium

ABSTRACT

The ongoing surge in side-channel attacks on trusted execution environments has prompted the application of formal methods to eliminate these vulnerabilities once and for all. Recently, research has focused on mitigating interrupt-driven side channels on small microcontrollers, resulting in a provably secure mechanism to maintain enclave isolation in the presence of arbitrary timer interrupts.

This paper uncovers a subtle flaw in the formal model's handling of illegal memory-access exceptions, showing how this oversight can compromise contextual equivalence through a strategic combination of interrupts and exceptions. We propose a minimal adjustment to rectify the model and reinstate formal assurances. Our study underscores the importance of precisely modeling interrupts and exceptions, drawing remarkable parallels with controlled-channel and single-stepping attacks on higher-end processors.

1 INTRODUCTION

Recent years have seen increasing adoption of trusted execution environments (TEEs) to safeguard sensitive data from privileged adversaries with full control over the operating system on the target device. This surge has, in turn, led to a long line of privileged side-channel attacks exploiting the TEE adversary's control over privileged processor features such as interrupts [10–12] and memory-access exceptions [1, 7, 13]. In response to these threats, various hardware-software co-designs have been proposed to mitigate these side channels, raising questions about how to assess the security assurances of these implementations. Focusing on interrupt-driven attacks on small microcontrollers, recent research has devised a promising mechanism [4, 5] to provably preserve enclave isolation in the presence of arbitrary timer interrupts, with practical implementation on the Sancus security architecture [8]. Subsequent studies [2, 3] have emphasized the importance of verifying the real-world implementation's alignment with the pen-and-paper formal model. Notably, to date no oversights have been identified in the (non-mechanised) proof logic itself.

Our research uncovers a subtle flaw in the formal model's handling of illegal memory-access exceptions. Specifically, we demonstrate how an attacker can break contextual equivalence by strategically combining interrupts and exceptions, mirroring tactics observed in combined page-fault [13] and single-stepping interrupt [6, 7, 11] attacks on real-world Intel SGX platforms. We propose a minimal adjustment to rectify the model and reinstate formal guarantees.

PAVeTrust'24, September 9, 2024, Milan, Italy
.

In conclusion, our study highlights the intricacies of formally modeling interrupts and exceptions and draws attention on the hurdles of formal proofs when not supported by tools (e.g., proof assistants).

2 FULL ABSTRACTION BREACH

In the original $Sancus_V$ papers [4, 5] the authors developed two formal models: Sancus^{H} reflecting the mental programming model of the programmer and the corresponding attacker and Sancus^{L} expressing the actual runtime model of $Sancus_V$ with interrupt-enabled attacker. These two models were then proved to be fully abstract [5, Theorem 6.3], meaning that any attack that can be performed in Sancus^{L} has a counterpart in Sancus^{H} and vice versa. More precisely, we say that two enclaves M and M' are *indistinguishable*, written $M \simeq^{\text{H}} M'$ (resp. $M \simeq^{\text{L}} M'$) for Sancus^{H} (resp. Sancus^{L}), whenever they equi-terminate in any context, i.e., a program with a hole acting as the attacker. Theorem 6.3 states that if a context C distinguishes two enclaves M and M' in Sancus^{L}, then it exists a context C distinguishing them in Sancus^{H} (and vice versa).

Investigation on $Sancus_V$ by Bognar et al. [2] uncovered vulnerabilities in the real-world Verilog implementation, while the formal models were apparently solid. Recently, Busi et al. [3] proposed ALVIE, a tool to automate the discovery of such implementation-model mismatches. ALVIE helped uncovering two new bugs in the $Sancus_V$ implementation that exploit the default behavior of resetting the CPU on memory-access violations.

Memory-Access Violations. To address the new bugs discovered by ALVIE and bring the $Sancus_V$ implementation in line with the formal model, Busi et al. [3] recommend that the CPU raises a dedicated exception that can be explicitly handled by both Sancus^{H} and Sancus^{L} adversaries. This can be achieved with a minimal change that explicitly clears the `RESET_ON_VIOLATION` variable (on by default) in the existing Verilog implementation of $Sancus_V$. In line with the formal model, whenever an instruction i violates memory-access permissions in enclave mode, a *separate*, attacker-controlled exception handler at `0xFFFE` is called at time $t + \text{cycles}(i)$ (while CPU registers and any pending timer interrupts are cleared).

Breaking Contextual Equivalence. We could break indistinguishability because the formal semantics overlooked the way it handles exceptions (e.g., memory violations). Indeed, we show a specific enclave pair that is indistinguishable to a Sancus^{H} adversary capable of observing exceptions but not triggering interrupts, yet distinguishable to a Sancus^{L} adversary who can additionally exploit interrupts. Consider the following minimal code snippet that branches based on an enclave-internal secret:

```
jz 1f; nop; nop; mov r8, &public # 1 + 1 + 4 cycles
1:     mov &private, &public     # 6 cycles
```

As the $Sancus_V$ semantics explicitly prohibits writing to public (unprotected) memory addresses outside the enclave, a memory-access exception is invariably triggered by the mov instruction. Note that, both in $Sancus_V$'s semantics and the Verilog implementation, exceptions are always deferred until instruction retirement. Consequently, enclave pairs containing an internal secret of either zero or non-zero will appear identical to a Sancus^{H} adversary lacking interrupt capabilities, who will consistently observe an identical exception occurrence in both branches at the same time (i.e., 6 cycles after the jz instruction). However, a Sancus^{L} adversary can easily differentiate between the two enclaves by strategically scheduling a timer interrupt immediately after the jz instruction and observing whether the associated interrupt handler gets called:

(1) if the secret is non-zero, the timer interrupt arrives during the 1-cycle nop instruction and the interrupt handler in the attacker context is invoked *before* any exception arises;
(2) if the secret is zero, the timer interrupt arrives during the 6-cycle mov instruction; however, upon instruction retirement, the pending timer interrupt is superseded by the memory-access violation and a *distinct* exception handler in the attacker's context is triggered.

The above behavior is formally modeled in the **Cpu-Violation-PM** and **Int-PM-P** rules of $Sancus_V$'s operational semantics. Additionally, we experimentally validated the contextual-equivalence breach in the publicly available Verilog implementation.

3 RECOVERING FULL ABSTRACTION

Recall that, the proof of full abstraction in Theorem 6.3 of [5] first shows that Sancus^{L} is backwards compatible with Sancus^{H} by proving *reflection of behaviors*, i.e., $\forall M, M'.\ M \simeq^{\text{L}} M' \Rightarrow M \simeq^{\text{H}} M'$. Then it proves that $M \simeq^{\text{H}} M' \Rightarrow M \simeq^{\text{L}} M'$ ($\star$), thus establishing *preservation of behaviors*, i.e., that the Sancus^{L} semantics is as secure as the original Sancus^{H}. This proof relies on traces – finite sequences of attacker-visible runtime actions – to show the contrapositive of ($\star$) by providing an algorithm that constructs a context in Sancus^{H} that distinguishes two enclaves when they are not trace equivalent (*backtranslation* [9]). Finally, the proof is completed by establishing that two modules that are trace equivalent are also indistinguishable in Sancus^{L}. However, the above counterexample requires to fix the model to recover full abstraction.

Dissecting the Counterexample. In the counterexample the two enclaves differ just by the secret and they execute as follows:

(1) If the secret is non-zero, the computation is $INIT_{s\neq 0} \rightarrow^*$ nop and then, by **Int-PM-P**, nop $\rightarrow$ IRQ where $INIT_{s\neq 0}$ is the initial state of the processor with the enclave and IRQ denotes the processor state at the beginning of the interrupt handling routine. The execution of nop gives the control back to the attacker. Depending on the nature of the attacker, the trace semantics of [5, Fig. 9] either records a $\bullet$ denoting convergence, or no observable if the attacker decides to diverge, or a $\texttt{jmpOut!}(\Delta t; \mathcal{R})$, for some timing Δt and register file $\mathcal{R}$, if the attacker resumes enclaves execution and waits for it to finish.
(2) If the secret is zero, the computation is $INIT_{s=0} \rightarrow^*$ mov &private, &public $\rightarrow$ EXC, where EXC denotes an exception state. In this case, since upon exception the control goes back to the attacker, the trace semantics emits $\texttt{jmpOut!}(\Delta t'; \mathcal{R}_0[\texttt{pc} \mapsto exc])$ where exc is the address of the exception handler. Here $\Delta t' = 2 + 6 = 8$ where $2 + 6$ is the sum of instructions timings inside the enclave. Note that here there is no padding for mitigation and that no setup time is needed for exceptions (see rule **CPU-Violation-PM**).

Harmonizing Exception Timing. The fix consists of harmonizing the management of time of rule CPU-Violation-PM (resp. **CPU-Violation-PM**) with that of CPU-Decode-Fail (resp. **CPU-Decode-Fail**), i.e., substituting $t + cycles(i)$ with t. To see why this works in general, consider any two enclaves M and M', distinguishable in Sancus^{L} by an interrupt happening at time t_{int} and such that (1) M has an instruction i starting at time t_{exc} causing an exception; and (2) M' has instruction i' starting at time t'_{exc} also causing an exception. With the old rule, in Sancus^{H} the exception handler starts at time $t_{exc} + cycles(i)$ for M and $t'_{exc} + cycles(i')$ for M', making the two enclaves indistinguishable in cases like ours. Instead, with the updated rule the exception handler starts at time t_{exc} for M and t'_{exc} for M' making the two cases always distinguishable because $t_{exc} \neq t'_{exc}$, otherwise both enclaves would have been interrupted at time t_{int} in Sancus^{L}, contrary to our assumption that one of them caused an exception.

With our proposed fix, the Sancus^{L} attacker can learn the secret by distinguishing[1] ISR from EXC, and the formal model correctly accounts for that. Since the Sancus^{H} attacker is built by backtranslation from the traces, it mimics the Sancus^{L} attacker but tries to learn the secret without the use of interrupts by letting the enclaves exit at their will. The minimal change in the semantics mentioned above, excludes the scenario of the counterexample and calls for minimal adjustments to the proof, in particular to the backtranslation. In detail, this means that both enclaves execute until rule CPU-Violation-PM fires and the attacker starts to execute code at address exc at time t if the secret is zero, t' otherwise. However, $t = t'$: if the secret is zero, the offending instruction is the 6-cycles mov &private, &public, otherwise the exception is caused by the 4-cycles mov r8, &public that follows 2, 1-cycle nops.

Implementation Considerations. While the proposed fix requires only minimal changes in the formal semantics, implementing our fix as-is in the real-world $Sancus_V$ processor would necessitate non-trivial modifications to the underlying openMSP430 pipeline design. Particularly, our proposal requires the CPU to detect exceptions *before* the offending instruction is actually executed, while the actual openMSP430 processor detects exceptions *during* attempted instruction execution (i.e., when the operands are known/computed) and delays handling them until instruction retirement [1, 12]. Thus, we propose a slight variation of our fix below that is semantic-preserving and straightforward to implement on the real-world $Sancus_V$ processor. It suffices to ensure that the time

[1] The attacker could distinguish the address of the interrupt-service routine (*isr*) from that of the exception handler (*exc*) or use timing to distinguish the two enclaves since it always takes Sancus 12 cycles to jump to *isr*, while the time between an exception and the jump to *exc* is ≤ 6, i.e., the duration of the longest Sancus instruction.

between the start of the offending instruction and that of the exception handler is a constant, e.g., MAX_TIME, which is already defined in the $Sancus_V$ implementation and equals the number of cycles taken by the longest instruction (i.e., 6 cycles).

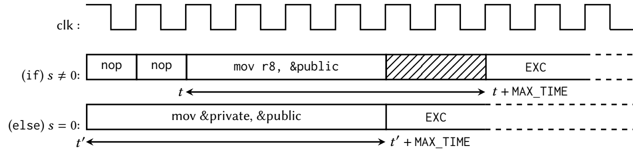

Figure 1: Graphical representation of an implementation of the proposed fix. The countermeasure re-uses existing Sancus mechanisms, in particular time padding, represented by the dashed rectangle.

Figure 1 illustrates a possible execution of our modified counterexample on a fixed $Sancus_V$ implementation. When the secret $s \neq 0$, the CPU executes two single-cycle `nop` instructions and starts executing `mov r8, &public` at time t, raising an exception during its execution. In this case, since `mov r8, &public` lasts 4 cycles, the CPU waits for MAX_TIME $- 4 = 2$ additional cycles (dashed rectangle in the figure) after the end of the offending instruction. Therefore, the exception handler starts at time $t +$ MAX_TIME. When $s = 0$, on the other hand, the exception is raised during the `mov &private, &public` instruction, that starts at time t'. In this case, the instruction lasts 6 cycles and the CPU needs not to wait additional cycles. The exception handler, therefore, starts at a *different* time $t' +$ MAX_TIME.

4 CONCLUSIONS AND INSIGHTS

The identified contextual-equivalence breach illustrates how TEE adversaries can exploit a strategic combination of side-channel information leakage from interrupts and exceptions to their advantage. Notably, while we are the first to study this amplification effect on small microcontrollers, similar tactics have previously been showcased on real-world Intel SGX enclave processors, where single-stepping frameworks like SGX-Step [10, 11] enable precise tracking of the number of enclave instructions leading up to an observable exception. The instruction-level temporal resolution provided by SGX-Step has proven particularly effective in enhancing the relatively coarse-grained (page-level) spatial resolution associated with attacker-induced exceptions like page faults [7, 13].

In a wider perspective, our discovery of a contextual-equivalence breach within the provably secure $Sancus_V$ architecture calls for tool-supported formal models and mechanized proofs and underscores the criticality of precisely modeling asynchronous processor features like interrupts and exceptions. We anticipate that our formalization can contribute to enhancing interrupt defenses in diverse TEE architectures.

ACKNOWLEDGEMENTS

This research was partially funded by the Research Foundation – Flanders (FWO) via grants #1261222N and #G081322N, the Research Fund KU Leuven, and the Flemish Research Programme Cybersecurity. Further support was provided by projects "SEcurity and RIghts In the CyberSpace - SERICS" (PE00000014 - CUP H73C2200089001), "Interconnected Nord-Est Innovation Ecoscheme - iNEST" (ECS00000043 - CUP H43C22000540006), and PRIN/PNRR "Automatic Modelling and ∀erification of Dedicated sEcUrity deviceS - AM∀DEUS" (P2022EPPHM - CUP H53D23008130001), all under the National Recovery and Resilience Plan (NRRP) funded by the European Union - NextGenerationEU.

REFERENCES

[1] Marton Bognar, Cas Magnus, Frank Piessens, and Jo Van Bulck. 2024. Intellectual Property Exposure: Subverting and Securing Intellectual Property Encapsulation in Texas Instruments Microcontrollers. In *33rd USENIX Security Symposium*.
[2] Marton Bognar, Jo Van Bulck, and Frank Piessens. 2022. Mind the Gap: Studying the Insecurity of Provably Secure Embedded Trusted Execution Architectures. In *43rd IEEE Symposium on Security and Privacy (S&P)*.
[3] M. Busi, R. Focardi, and F. Luccio. 2024. Bridging the Gap: Automated Analysis of Sancus. In *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*. 347–362.
[4] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. 2020. Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors. In *33rd IEEE Computer Security Foundations Symposium (CSF)*.
[5] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. 2021. Securing Interruptible Enclaved Execution on Small Microprocessors. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43, 3 (2021), 1–77.
[6] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. 2023. AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves. In *32nd USENIX Security Symposium*. 4051–4068.
[7] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. 2020. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *29th USENIX Security Symposium*. 469–486.
[8] J. Noorman, J. Van Bulck, J. Tobias Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. 2017. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Transactions on Privacy and Security* 20, 3 (2017), 1–33.
[9] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst.* 37, 2 (2015), 6:1–6:50.
[10] Jo Van Bulck and Frank Piessens. 2023. SGX-Step: An Open-Source Framework for Precise Dissection and Practical Exploitation of Intel SGX Enclaves. In *ACSAC 2023 Cybersecurity Artifacts Competition and Impact Award Finalist Short Paper*.
[11] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *2nd Workshop on System Software for Trusted Execution (SysTEX)* . ACM, 4:1–4:6.
[12] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *25th ACM Conference on Computer and Communications Security (CCS)*. 178–195.
[13] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *36th IEEE Symposium on Security and Privacy (S&P)*. 640–656.